Inheritance basics

Principle:
Describe a new class as extension or specialization of an existing class (or several with multiple inheritance)

If \( B \) inherits from \( A \):
- Module viewpoint: all the services of \( A \) are available in \( B \) (possibly with a different implementation).
- Type viewpoint: whenever an instance of \( A \) is required, an instance of \( B \) will be acceptable ("is-a" relationship)

Terminology

Parent, Heir
Ancestor, Descendant
The ancestors of \( B \) are \( B \) itself and the ancestors of its parents.
Proper ancestor, Proper descendant
Direct instance, Instance
The instances of \( A \) are the direct instances of its descendants.
(Other terminology: subclass, superclass, base class)

Example hierarchy

Redefinition 1: polygons

```plaintext
class POLYGON inherit CLOSED FIGURE
create
make
feature
vertex : ARRAY [POINT]
vertex_count : INTEGER
perimeter : REAL
is
-- Perimeter length
do from ... until ... loop
Result := Result + vertex[i].distance(vertex[i+1])
end
invariant
vertex_count >= 3
vertex_count = vertex.count
define deferred
-- effective
++ redefined
vertex
```

Intro. to Programming, Lecture 13: Inheritance & Genericity

Einführung in die Programmierung
Introduction to Programming
Prof. Dr. Bertrand Meyer
October 2006 – February 2007

Lecture 13: Inheritance & Genericity
Redefinition 2: rectangles

class RECTANGLE inherit POLYGON
  redefine perimeter
end

create make

feature
  diagonal, side1, side2 : REAL
  perimeter : REAL
  -- Perimeter length
  do Result := 2 * (side1 + side2)
end

invariant
  vertex_count = 4
end

Inheritance, typing and polymorphism

Assume:
  p: POLYGON ; r: RECTANGLE ; t: TRIANGLE,
  x: REAL

Permitted:
  x := p.perimeter
  x := r.perimeter
  x := r.diagonal
  p := r

NOT permitted:
  x := p.diagonal -- Even just after p := r !
  r := p

Dynamic binding

What is the effect of the following (assuming some Test true)?

  if some Test then
    p := r
  else
    p := t
  end

  x := p.perimeter

Redefinition: A class may change an inherited feature, as with
POLYGON redefining perimeter.
Polymorphism: p may have different forms at run-time.
Dynamic binding: Effect of p.perimeter depends on run-time form of p.

Without inheritance!

display (f : FIGURE ) is
  do
    if "f is a CIRCLE" then
      ...
    elseif "f is a POLYGON" then
      ...
    end
  end

and similarly for all other routines!

Tedious; must be changed whenever there's a new figure type.

With inheritance!

With:

  f: FIGURE
  c: CIRCLE
  p: POLYGON

and:

  create c.make ( ... )
  create p.make ( ... )

Then just use:

  f.move ( ... )
  f.rotate ( ... )
  f.display ( ... )
  -- and so on for every
  -- operation on f !

Inheritance: summary 1

- Type mechanism: lets you organize our data abstractions into taxonomies
- Module mechanism: lets you build new classes as extensions of existing ones
- Polymorphism: Flexibility with type safety
- Dynamic binding: automatic adaptation of operation to target, for more modular software architectures
### Genericity

**Unconstrained**

\[ \text{LIST}[G] \]
- e.g. LIST[INTEGER], LIST[PERSON]

**Constrained**

\[ \text{HASH\_TABLE}[G \rightarrow \text{HASHABLE}] \]
- \[ \text{VECTOR}[G \rightarrow \text{NUMERIC}] \]

### Extending the basic notion of class

#### Abstraction

- Set of cities
- List of cars
- Linked list of cars

#### Inheritance

- Abstraction
- Specialization

#### Generics

- Type parameterization
- Constrained

### Extending the basic notion of class

#### Inheritance

- Abstract concept
- Specialization

#### Generics

- Type parameterization
- Constrained

### Genericity: Ensuring type safety

How can we define consistent "container" data structures, e.g. list of accounts, list of points?

**Dubious use of a container data structure:**

```plaintext
How if wrong?
c : CITY; p : PERSON
cities : LIST ...
people : LIST ...
--------------------------------------------------------
people.extend ()
cities.extend ( )
c := cities.last
```

**What if wrong?**

```plaintext
p.c.add_tram_line (Line8)
```

### Possible approaches

1. Duplicate code, manually or with help of macro processor.
2. Wait until run time; if types don’t match, trigger a runtime failure. (Smalltalk)
3. Convert (“cast”) all values to a universal type, such as “pointer to void” in C.
4. Parameterize the class, giving an explicit name \( G \) to the type of container elements. This is the Eiffel approach, now also found in Java, .NET and others.

### A generic class

```plaintext
class LIST[G] feature
extend(x: G) is ...
last: G is ...
end
```

To use the class: obtain a generic derivation, e.g.

```plaintext
cities: LIST[CITY]
```
Using generic derivations

```plaintext
cities : LIST[CITY]
people : LIST[PERSON]
c : CITY
p : PERSON
...
cities.extend (c)
people.extend (p)
c := cities.last
c.add_tram_line (Line8)
```

Static typing

**Type-safe call** (during execution):

A feature call `x.f` such that the object attached to `x` has a feature corresponding to `f`. [Generalized to calls with arguments, `x.f(a, b)`]

**Static type checker**:

A program-processing tool (such as a compiler) that guarantees, for any program it accepts, that any call in any execution will be type-safe.

**Statically typed language**:

A programming language for which it is possible to write a static type checker.

The static and the dynamic

For a feature call `x.f`:

**Static typing**:

There is at least one feature `f` applicable to `x`

**Dynamic binding**:

If more than one possible feature, execution will select the right feature

Using genericity

```plaintext
LIST[CITY]
LIST[LIST[CITY]]
...
```

A type is no longer exactly the same thing as a class!

(But every type remains based on a class.)

What we have seen

- Inheritance
- Polymorphism
- Dynamic binding
- Static typing
- Genericity
End of lecture 13